**Output from running git pull --rebase, git push, git status, git rev-parse HEAD**

[michaelgreen1@onyxnode22 p1-shell\_part1]$ git status

# On branch master

nothing to commit, working directory clean

[michaelgreen1@onyxnode22 p1-shell\_part1]$ git pull --rebase

Current branch master is up to date.

[michaelgreen1@onyxnode22 p1-shell\_part1]$ git push

Everything up-to-date

[michaelgreen1@onyxnode22 p1-shell\_part1]$ git rev-parse HEAD

6015edbedc910de58726bd55dc358c8bdde8a7c7

**Output from running make clean, make**

[michaelgreen1@onyxnode22 p1-shell\_part1]$ make clean

/bin/rm -f \*.o mydash a.out core  \*.d version.c

[michaelgreen1@onyxnode22 p1-shell\_part1]$ make

gcc -g -O -MMD -Werror -Wall -Wextra -std=c89 -pedantic -Wmissing-prototypes -Wstrict-prototypes -Wold-style-definition   -c -o mydash.o mydash.c

gcc -g -O -MMD -Werror -Wall -Wextra -std=c89 -pedantic -Wmissing-prototypes -Wstrict-prototypes -Wold-style-definition   -c -o log.o log.c

echo -ne '#include "version.h"\n' > version.c

echo -n 'const char\* version(void) { const char\* GIT\_Version = "' >> version.c

git rev-parse HEAD | cut -c1-6 | tr -d '\n'  >> version.c

echo '"; return GIT\_Version; }' >> version.c

gcc -g -O -MMD -Werror -Wall -Wextra -std=c89 -pedantic -Wmissing-prototypes -Wstrict-prototypes -Wold-style-definition   -c -o version.o version.c

g++ -g -Wall -Werror   -c -o list.o list.cpp

g++ -g -Wall -Werror -o mydash  mydash.o log.o version.o list.o -lreadline -lncurses

[michaelgreen1@onyxnode22 p1-shell\_part1]$

**Background jobs 10/10**

*The first step to this assignment was to get programs to run in the background. So I started by identifying the ampersand, which was a trial under itself. Then I set a Boolean that made sure the appropriate strings were not freed after I ran through the exec process, and then I added the job to the list. Hard to confirm without the job command, so I implemented that next. Not sure if that fell under the simple job control or not. I also had to go back and add another method to make sure that the jobs were being stored at the correct index. I believed that indexes reset, but they are static and based on current job ordering. Also, I used the list as a stack, did not add code to the C++. Missed that day of class, otherwise probably would have, with further explanation. However, it isn’t too difficult to turn two stacks into a queue, and it makes the run time 2n rather than n. Not that bad considering how large job lists are in the shell. Would be a problem on kernel level, though.*

**History 5/5**

*This took a bit of effort with readline, about a hour of reading through documentation before I figured out how to get the double array of history\_structs and then accessing the line information from those structs. Also figuring out how to handle null entries into the history. That line of code took a minute to process.*

**Simple Job control 15/15**

*Job handling. So essentially I hacked this. I found out later that there was code on piazza and I understand that waitpid gives the correct status if you do this correctly. But instead I added a variable to my job struct, a Boolean that says whether or not the job is a stopped. This is really all we are checking in this iteration of smash, so it worked just fine. Took me a minute to figure out how to turn two stacks into a queue, and of course output formatting is a chore, but not overly difficult. Up until this point the programming was going well.*

**Signal handling 25/25**

*Hellish. Nah not that bad, but I had no clue what I was doing for a really long time, and then something finally clicked and I understood how it worked. How separate process groups handle signals differently than if they were in the same process group as their parents. This tied into recursive invocation. I kept on messing around with setting who had the terminal, and for a while I couldn’t get signal handling and recursive invocation to work. But I figured out where I was making the mistake. Funny how worked it got me, and how simple it seems afterwards. For a while I Was using signal handlers that set global variables which allowed me to see if a job had been stopped or terminated. Potentially brilliant, if not entirely unncessarry. Reinventing the wheel. It was fun though, breaking the rules to make it all work. However, I Could not keep this build because the signal handling prevented terminal control in the recursive invocation. And once I had figured out how to handle the signals, there was no real need to keep the global variables.*

**Recursive Invocation 10/10**

*Once I understood signals, and terminal control, recursive invocation was a cinch. It was just a matter of setting terminal control, and appropriate process grouping.*

**Additional job control features 25/25**

*Not too difficult as well, once signal handling is understood. It was a bit difficult to be able to pull out jobs at a specific point in the stack, well, pulling them out wasn’t difficult. Adding them back at the right place required a bit of thinking however, and an extra method addAt. A bit of fooling around with the handling of a command that can be with or without an extra modifier. Atoi() is still a bit of a mystery to me, and the man page is terse. What happens when I use atoi on a non-numeric string? Returns 0? I got it to work, but managing it is kind of unpleasant. I might add a bit more error handling in this section after I hand this in.*

**Valgrind output 10/10**

*Followed the suggested pattern of implementing a feature and then running it through valgrind. This kept the memory leaks to a minimum. I had a few that took a while to hunt down, but memory handling is also pretty intuitive with practice. If it is allocated, free it. But only after it is done.*

**Total Grade 100/100**

*I enjoyed this project a lot, It was incredibly challenging for me, and for a moment I wasn’t sure if I would be able to do it. Frustrating that I had to miss the class and code through the flu. Probably would have been easier if that hadn’t been the case. But, I suppose it is very heartening as well, to be capable of finishing this project despite my miserable date and desperate pilfering of man pages. This project was incredibly hard, and I loved it for that. More so in retrospect. When I calmed down and thought through it, it made sense.*

*With this project I took to heart what you said. “Understand what the code does”. That’s what I wanted to do, and still continue to want. The extra week really helped, allowed me to push this from barely working B grade code, to relatively polished and understood code.*